ARCHITECTURAL SUPPORT FOR WORK-EFFICIENT RELAXED PRIORITY QUEUEING

BY

AZIN HEIDARSHENAS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Josep Torrellas

ABSTRACT

Many parallel algorithms in domains such as graph analytics and simulations execute more efficiently if some parallel tasks are executed before others. To implement such priority-based task scheduling, the data structure of choice is a concurrent priority queue (PQ). Unfortunately, PQ algorithms exhibit an undesirable tradeoff. On one hand, traditional PQs always dequeue the highest-priority task, and thus fail to scale because of contention at the head of the queue. On the other hand, relaxed PQs avoid contention by dequeuing tasks that are often so far from the head that the resulting schedule misses the benefit of priority-based scheduling.

This thesis proposes novel architectural support for relaxing PQs without straying far from the priority-based schedule. Our architecture, called Snug, distributes the PQ and maintains a set of *Work Registers* that point to the highest-priority task in each subqueue. Snug provides an instruction that picks a high-quality task to execute. The instruction periodically switches between visiting all the subqueues to get an accurate global snapshot and visiting nearby subqueues to reduce traffic. Overall, Snug dequeues high-quality tasks while simultaneously avoiding hotspots and excessive network traffic. We evaluate Snug on graph analytics and event simulation applications. Snug reduces the average execution time of the applications by 1.6×, 4.9× and 3.4× compared to the state-of-the-art skip list, SprayList, and software-distributed PQs, respectively.

To my parents and friends, for their love and support.

ACKNOWLEDGMENTS

My sincerest thanks are extended to my thesis advisor Prof. Josep Torrellas for his priceless support and constant guidance, Prof. Adam Morrison, of the University of Tel-Aviv, for his brilliance and brainstorming ideas, and my colleague Tanmay Gangwani for his expertise and incredible patience to teach me. I am extremely grateful that they considered me as as part of this project.

I am also incredibly indebted to my professors, labmates, classmates, TAs and virtually anyone who I learned from so far here. I am extremely thankful to be part of one of the most world-renowned engineering college communities.

Thank you my wonderful friends: Parisa, Mona, Tahere, Amir, Hadi, Nikta, Fardin, Alia, Farnaz, Setare, Bardia, Amir Hossein, Ishan, Aakash, Ashok, and Jose. You should know that your presence in UIUC has been worth more than I can express on paper.

Last but not least, I would like to thank my three best friends for their invaluable emotional support and encouragements throughout my entire life: My mom, who has taught me to never give up pursuing my dreams, my dad, who was the first person to teach me what it means to be an engineer, and my lovely brother. Thank you from the bottom of my heart!

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Many parallel algorithms in domains such as graph analytics [1, 2, 3], discrete event simulation [4], and SAT solving [5] execute more efficiently if some parallel tasks are executed before others. In this case, the desired execution order can be attained with *priority-based* task scheduling: when the algorithm creates a task T, it assigns a priority p to T, and if $T_1.p < T_2.p$, then T_1 should execute before T_2. (That is, lower p values mean higher priorities.) The data structure of choice for implementing priority-based task scheduling is a concurrent priority queue (PQ). A PQ maintains a collection of items, each associated with a priority. It supports two basic operations: *enqueue*, which adds an item to the correct position in the queue, and *dequeue*, which removes the item with the highest priority from the head of the queue and returns it.

Unfortunately, concurrent PQs are not scalable. Since every thread executing dequeue tries to remove the same highest-priority item, the head becomes a synchronization hotspot [6, 7, 8, 9, 10, 11]. The resulting serialization and synchronization overheads can dominate the execution time.

To avoid this problem, researchers have proposed to *relax* PQ semantics and allow dequeue to return an item that is not the one of highest-priority [12, 13, 14, 15]. Relaxed PQ algorithms use a variety of strategies to find the item to dequeue. For example, some perform a short random walk on a skip list to find an item to dequeue [12], while others pick the highest-priority item between a thread-local PQ and a global shared PQ [14]. These strategies alleviate the bottleneck at the head of the queue.

Relaxed PQ algorithms face the danger of straying too far from the desired task execution order and ending up performing wasted work. For example, in discrete event simulation, a thread may process an event that is far in the future, only needing to reprocess it again later as new events that occur from now until that time change the system state. Thus, the key difficulty

in designing a relaxed PQ is to return high-priority items, i.e., those that are close to the PQ head.

In practice, existing relaxed PQs often fail to achieve this goal. For example, the SprayList [12] returns an item among the first $O(t \log^3 t)$ ones in the PQ with high probability, where t is the number of threads. For a 64-thread execution, this translates to a weak guarantee of returning an item within the first $13,824$ in the queue (ignoring constant factors). Similarly, with the recommended parameter $k = 256$, the k-LSM PQ [14] only guarantees returning an item within the first $16,384$ in the queue in a 64-thread execution. Also, how well relaxed PQs distribute synchronization, as well as the quality of the relaxation, is controlled by (sometimes multiple) parameter knobs [12, 13, 14]. Determining the right values of these knobs is challenging, since they depend on many factors. Overall, current priority-based task scheduling algorithms pose an unfortunate synchronization vs. work-efficiency trade-off: alleviating the synchronization hotspot by using relaxed PQs leads to wasted work.

This thesis introduces novel hardware support to avoid the above-mentioned trade-off. Our architecture, called Snug, relaxes the priority order in a PQ algorithm slightly, to alleviate contention while inducing, on average, little wasted work. Snug distributes the PQ and maintains a set of *Work Registers* that point to the highest-priority task in each subqueue. A new *PickHead* instruction returns a high-priority task from the combined PQ for processing.

PickHead employs multiple techniques to pick a high-priority task without congesting the network. Initially, *PickHead* reads all *Work Registers* in parallel and returns a random task from within the R highest-priority ones observed. R is called the *Relaxation Count*, and is dynamically adapted by Snug, based on the rate of synchronization failures—which indicate the degree of contention. In later *PickHead* invocations, *PickHead* reuses the obtained information to avoid rereading the *Work Registers*. Finally, *PickHead* sometimes reads from a set of nearby *Work Registers*, instead of from all the registers, to save traffic. Overall, *PickHead* performs high-quality task selection while avoiding hotspots, minimizing wasted work, and consuming acceptable network bandwidth.

Snug effectiveness. We evaluate Snug on a simulated 64-core chip augmented with Snug hardware, using graph and discrete event simulation ap-

plications with various inputs. We compare the execution time of the applications using SNUG and using several other PQ algorithms. Such algorithms include software-only PQs based on a skip list, a SprayList, and a distributed skip list; and a hardware-based centralized PQ. SNUG reduces the average execution time of the applications by 1.6×, 4.9× and 3.4× compared to the state-of-the-art skip list, SprayList, and software-distributed PQs, respectively. Compared to the latter two relaxed PQ designs, SNUG reduces the number of wasted tasks by 4.4× and 18.2×, respectively.

Overall, the contributions of this thesis are:

• The SNUG architecture, consisting of *Work Registers*, a *PickHead* instruction and other Instruction-Set Architecture (ISA) extensions, and a design for dynamically adapting the relaxation count to minimize PQ contention.

• Simulation-based evaluation of SNUG using graph and discrete event simulation applications, and a comparison to several other software- and hardware-based PQs.

CHAPTER 2

BACKGROUND

2.1 Need for Priority-Based Task Scheduling

Many parallel algorithms share a similar structure. The work is decomposed into *tasks*, which can generate new tasks as they run. The algorithm remains correct regardless of the task execution order—which enables parallel task execution—but executes more efficiently if some tasks execute before others, according to some notion of priority. Processing a task out of priority order leads to wasted work, where a task executes inefficiently and/or the results of its computation are thrown away later. Examples of such algorithms include graph algorithms, such as single-source shortest path (SSSP) [1, 2], minimal spanning tree (MST) [16, 3], and betweenness-centrality [17]; discrete event simulation [4]; and SAT solving [5].

The SSSP example. Given a weighted directed graph and a *source* node s, SSSP finds the weight of the shortest path from s to every other node in the graph. Most SSSP algorithms use *relaxations* [18], in which the algorithm tests whether a shortest path found so far can be improved. Each node v is associated with a $dist(v)$ label (initially 0 for s and ∞ for all other nodes). A relaxation considers an edge (u, v) of weight w. If $dist(u) + w < dist(v)$, the algorithm updates $dist(v)$ to be $dist(u) + w$. Updates of a node's label are synchronized (e.g., with atomic instructions), allowing relaxations to run in parallel. The algorithm thus converges to the labels containing the weights of the shortest paths from s.

Notice that any relaxation that does not update a node's label to its true distance can be considered wasted work, since it will be overwritten by the relaxation that updates the label to the true distance. Dijkstra's SSSP algorithm [18, 1] relaxes each edge exactly once. It partitions the graph into

explored nodes, whose distance from s is known, and unexplored nodes. In each iteration, it picks the unexplored node u with the smallest label, marks it explored, and relaxes every edge (u, v).

Priority-based task scheduling. By using node labels as *priorities* and defining a task as relaxing every outgoing edge of a node, we can approximate Dijkstra's algorithm and minimize wasted work. Note that due to parallelism, wasted work is not guaranteed to disappear. This is because two tasks may perform conflicting relaxations. Empirically, however, this does not occur often as discussed in Chapter 5. Similar concerns exist in other algorithms, such as those mentioned above.

2.2 Concurrent Priority Queue Algorithms

Concurrent Priority Queues (PQs) are the natural data structures for implementing priority-based task scheduling. A PQ maintains a collection of items (i.e., tasks or nodes), each associated with its own priority p. We consider lower p values to mean higher priorities. A PQ supports two basic operations: (1) *enqueue* adds an item to the collection, and (2) *dequeue* removes the item with the highest priority (smallest p value) from the collection and returns it.

PQs suffer from scalability problems, due to synchronization hotspots resulting from the fact that every thread executing dequeue tries to remove the same, highest-priority item. Motivated by this scalability problem, researchers have *relaxed* PQs, allowing a dequeue to return a task that is not the highest-priority one [12, 13, 14, 15]. Relaxed PQs alleviate the synchronization hotspot, but increase the amount of wasted work. As a result, the performance of the application may improve or degrade with a relaxed PQ.

Contention in standard PQs. Modern PQs [6, 19, 8, 10, 11] are implemented based on the *skip list* data structure [20]. A skip list is conceptually a sorted linked list in which some nodes contain "hints" that enable searching in logarithmic time. PQs implement enqueue by inserting an item into the skip list (which is sorted by priority), and dequeue by removing the head item. For simplicity, we explain the PQ synchronization issues using the

simpler *linked list*-based PQ, and discuss skip lists in Appendix B.

Linked-list insertions are performed by searching the sorted list for the correct place to insert the new item, and linking a new node between the successor and predecessor found by the search. Searching is done using only reads, without acquiring locks [21]. The result is that enqueue operations can proceed in parallel and scale well.

Linked-list deletions are done in two phases: the node is first *logically* deleted by setting a "deleted" flag in the node, and then *physically* deleted by updating the *next* pointer of its predecessor. We explain the details in Appendix A. Both of these steps involve synchronization, either through locks or atomic Compare-And-Swap (CAS) instructions. Crucially, threads concurrently performing an enqueue all attempt to delete the same node, resulting in a significant synchronization bottleneck. PQ research has focused on reducing the impact of this bottleneck—e.g., by batching physical deletions [8]—but inevitably hits a scalability limit.

Relaxed PQs. Relaxed PQs eliminate the bottleneck at the PQ head by distributing dequeue operations, often using randomization. SprayList [12] is based on a centralized skip list, but a dequeue chooses its target item by performing a short random walk on the list. MultiQueues [13] and Sheaps [22, 23] distribute the data structure, composing the logical PQ out of per-thread PQs. Dequeues are performed from a remote queue only if the local queue is empty [23], or from a queue chosen with a randomized protocol [13]. Other approaches combine per-thread PQs with a global centralized PQ [14].

Overall, relaxed PQ designs trade off synchronization costs with increased probability of wasting work, since the tasks returned by a relaxed PQ may be far from the highest-priority tasks. For example, the SprayList [12] only guarantees (with high probability) to return an element among the first $O(t \log^3 t)$ in the PQ, where t is the number of threads. The synchronization vs. relaxation trade-off is sometimes controlled by multiple parameter knobs [12, 13, 14]. Tuning these knobs poses a challenge, as the right value depends on the number of threads, the input data, and the properties of the parallel algorithm being used [12, 13, 14].

CHAPTER 3

SNUG ARCHITECTURE

3.1 Main Idea

The goal of this thesis is to design a PQ that minimizes both wasted work and synchronization overhead. We accomplish both effects with SNUG, which is a novel architecture for high-performance, scalable, distributed PQs.

SNUG exists in the context of a large cache-coherent multi-core with a distributed directory. Each core (or core cluster) is physically close to a module of the distributed directory. A single, programmer-declared *logical* queue is distributed into multiple *physical* queues with their heads in the directory controllers.

In such an environment, when a thread running on a core calls the *enqueue* operation, the PQ library leverages the SNUG hardware to *enqueue* the node at the correct spot in the core's local physical queue. When the thread calls the *dequeue* operation, the PQ library leverages the SNUG support to return a node with one of the highest priorities in the PQ to the thread. The hardware inspects a strategic set of the queues, to minimize both wasted work and synchronization overhead and to avoid excessive traffic. Overall, with SNUG, enqueues are fast because they are local, while dequeues are both fast and return high-priority nodes thanks to special hardware.

Figure 3.1 shows the SNUG architecture. In a tiled multi-core, each node has two hardware structures: a set of *Work Registers* in the directory controller, and a *PickHead* module in the core. Each *Work Register* can function as the head of a work queue. Cores can access all *Work Registers* in the chip as memory-mapped locations in an uncacheable virtual address range. When a *Work Register* is used, it has two 8-byte values: a pointer to the first node in its queue, and that node's priority. Such a design enables any core in the chip to read the head of any of the physical queues without causing a

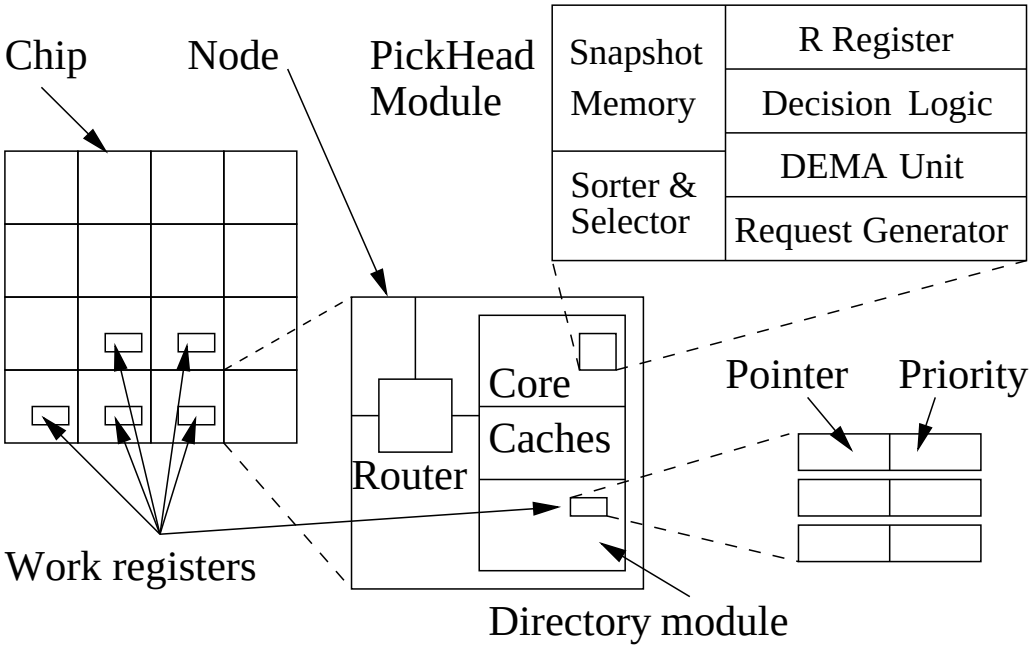


Figure 3.1: SNUG architecture in a directory-based multi-core.

ping-pong effect of the cached data across the network. In addition, there are instructions that read both pointer and priority together, and that also modify both pointer and priority atomically.

When a thread calls the dequeue operation, the PQ library issues a new instruction called *PickHead*. The instruction may access all the queues, a set of nearby queues (perhaps only the local queue), or no queue at all. For each of the queues visited, the network transaction returns the information in the *Work Register*—i.e., a pointer to the node at the head of the queue and its priority.

When all or some of the queues are accessed, the returning information is stored in a small *Snapshot Memory* in the core. *PickHead* then provides the information for one of the highest-priority nodes in there to the software. Sometimes, however, *PickHead* may not issue any network request at all. In that case, it simply reuses information in the *Snapshot Memory*, returning information for one of the nodes already there. In all cases, the library then attempts to dequeue that node. If it fails, it calls *PickHead* again, and the process repeats until a node is successfully dequeued.

The *PickHead* module in Figure 3.1 supports the *PickHead* instruction. *PickHead* has low overhead because, if the instruction visits any queues, it

triggers parallel transactions that read each of the relevant *Work Registers* in parallel. Moreover, it minimizes wasted work because the nodes obtained are the highest-priority nodes of the queues visited. However, we do not want to return the very highest-priority node in the PQ because it would cause all the threads to attempt a CAS on the same node. Instead, to reduce contention, we use three mechanisms as follows:

(1) SNUG sorts all the nodes in the *Snapshot Memory* based on their priority, considers only the top R of them, and picks one node at random to return to the software. R (*Relaxation Count*) is stored in the *R Register*, and is dynamically adapted by SNUG based on the rate of synchronization failures. (2) *PickHead* sometimes chooses to visit no queue and, instead, returns another entry from the *Snapshot Memory*. (3) Once multiple entries from the *Snapshot Memory* have been returned, *PickHead* chooses to visit only the local queue (or a set of nearby queues) for a few times, rather than all the queues.

The last two relaxations trigger inexpensive transactions. They induce minimal or no traffic and contention, while often providing high-priority nodes to dequeue.

3.2 *PickHead* Module Architecture

The *PickHead* module shown in Figure 3.2 implements the *PickHead* instruction. The goal of the instruction is to return one of the highest-priority nodes in the PQ with minimal overhead. As the instruction starts executing, a Decision Logic module makes one of three choices: (1) issue a global access to all *Work Registers*, (2) issue a local access, or (3) issue no network access at all.

Global access. Under certain conditions, *PickHead* issues as many network transactions as there are physical queues in the requested virtual queue. These transactions proceed in parallel, each visiting a queue and returning a 3-tuple to the *Snapshot Memory* (Figure 3.2). A tuple is composed of the virtual address (VA) of the queue (which we call the queue ID) and the contents of its Work register—i.e., a pointer to the node at the head of the queue, and the node's priority. At the end, *PickHead* needs to return one of these nodes to the software, which will then perform a CAS to attempt to

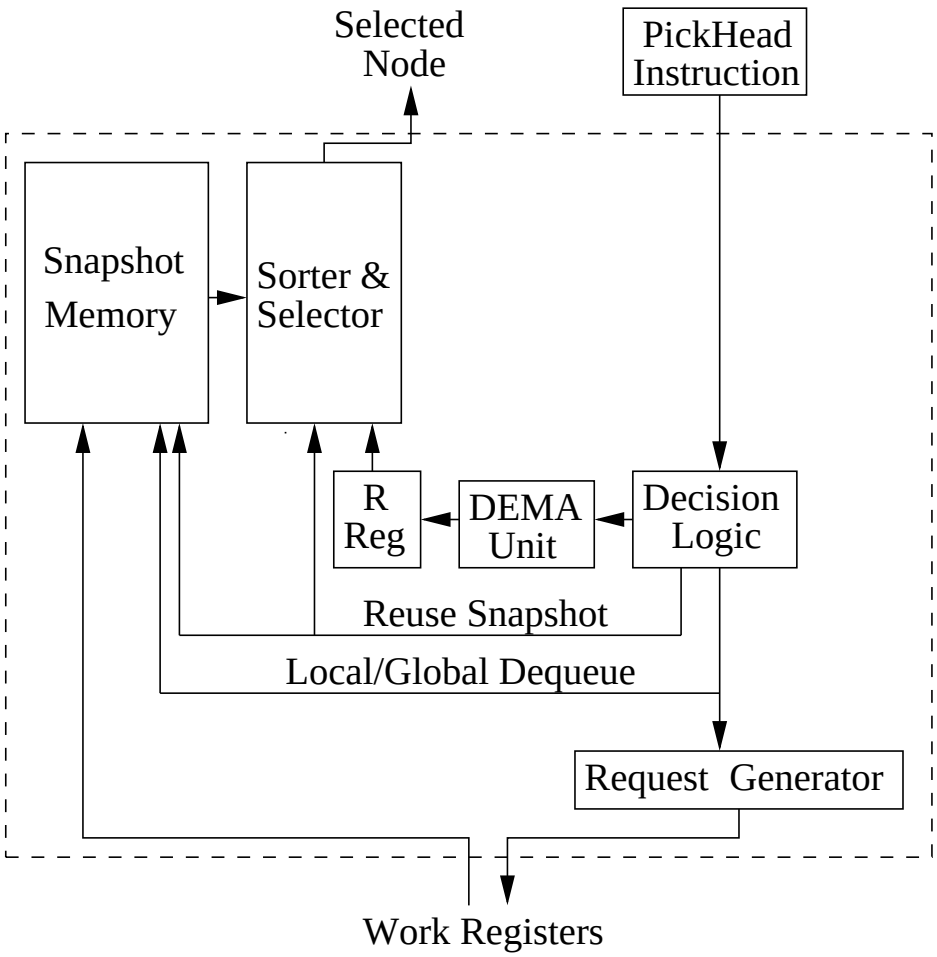


Figure 3.2: *PickHead* module.

dequeue the node from its corresponding queue.

If the chosen node was always the highest-priority node, we would induce high PQ contention. Therefore, the Sorter and Selector module (Figure 3.2) first sorts the nodes in the *Snapshot Memory* in decreasing priority, and then selects the subset of the nodes that have the highest priorities. This subset is called the *Inclusion Set*. Its size is equal to the *Relaxation Count* stored in the *R Register* (Figure 3.2). Finally, the *PickHead* module randomly selects one of the nodes in the Inclusion Set and returns it to the software.

We will describe the sorting step in Section 4.2. Once the nodes are sorted, note that there are many possible algorithms for picking a node from the Inclusion Set—e.g., use a function of the core ID.

To set R, we would ideally use feedback from the contention for the PQ and the amount of wasted work performed by the application. Unfortunately,

wasted work is very application dependent, and programmers may find it difficult to estimate. On the other hand, PQ contention is easy to measure. Hence, in SNUG, the *PickHead* instruction takes an operand (*PrevFailed?*) that indicates whether the CAS on the node provided by *PickHead* in the previous global access failed. The value of this operand is generated inside the PQ dequeue library. In general, if the frequency of failures of CAS operations is high, we want R to increase; if it is low, we want R to decrease. The Double Exponential Moving Average (DEMA) unit (Figure 3.2) uses the CAS success/failure history to set R. Section 4.2.1 explains the algorithm used.

Local access. Under certain conditions, *PickHead* accesses only the local queue or a few nearby queues. If the network is organized in clusters, *PickHead* could access all the queues in the local cluster. This transaction is inexpensive in terms of traffic and contention, but can potentially obtain a node with a slightly-lower priority than if we performed a global access.

This case proceeds as explained for the global access. Specifically, the responses are received in the *Snapshot Memory*; then, the Sorter and Selector sorts the priorities of the nodes and randomly picks one node from the top R highest-priority nodes—or from the total nodes, if there are fewer nodes than R. In the corner case when *PickHead* accessed a single queue, the returning data bypasses the *Snapshot Memory* and Sorter and Selector, and is returned directly to the software.

As a balance between the desire to reduce the global traffic and to pick high-priority nodes, we design *PickHead* to perform a fixed number of local accesses between any pair of consecutive global accesses. Section 4.1 explains the algorithm used to interleave global and local accesses.

No network access. After a *PickHead* instruction that performed a global access, several subsequent *PickHead* invocations do not access the network at all. Instead, they reuse the information that is currently stored in the *Snapshot Memory*. The goal is to reduce the network traffic without diminishing the quality of node selection much.

Specifically, recall that a global access brings information on one node for each of the physical queues. Such information is sorted and one of the top R nodes is returned to the software. The information for such node in the Sorter and Selector module is marked as consumed. Then, a subsequent *PickHead* invocation reuses the sorted array of nodes in the Sorter and Selector module

as follows. The hardware re-considers all the non-consumed entries in the module, and randomly selects one to return to the software. That entry is then marked as consumed. Note that the algorithm is otherwise not selective in what nodes to re-consider. The reason is to maximize the chances of attaining an available node. This process is repeated several times before the snapshot is discarded. Algorithm details are shown in Section 4.1.

3.3 Interface to the Software

The SNUG hardware is accessible with the four instructions in Table 3.1. The first one is *AllocHeads*, which allocates one or more *Work Registers* to be queue heads. *AllocHeads* takes two input operands and one output operand. The inputs are the number of programmer-visible logical queues to allocate, and the number of physical queues per logical queue. The physical queues of each logical queue are distributed into as many different directory controllers as possible. If the second input is zero, all the directory controllers in the chip receive a physical queue for each of the logical queues.

Table 3.1: SNUG instructions.

AllocHeads input: Number of logical queues input: Number of physical queues per logical queue output: VAs of all the physical queues allocated
PickHead input: ID of the logical queue input: Did the CAS on previous *PickHead* node failed? output: VA of the chosen physical queue to dequeue from output: Pointer to the (expected) first node in chosen queue
UpdateHead input: VA of the physical queue input: Pointer to the node at the head of the physical queue input: Pointer to the node to place at the queue's head input: Priority of the node to place at the queue's head output: Successful or failed outcome
FetchHead input: VA of the physical queue output: Pointer to the node at the head of the physical queue

AllocHeads returns the virtual addresses (VAs) of all the physical queues allocated—i.e., the VAs of all the *Work Registers* allocated. For example,

if the instruction allocates N logical queues with M physical queues each, it returns a matrix with $N \times M$ VAs.

```
1  void enqueue( int LogQueID, Node *new) {
2      Node **head = &queue[LogQueID][local]; // local queue
3      // head is the uncacheable address of the head of the queue
4      Node **prev = head;
5      Node *curr = FetchHead (head); // get node at head of queue
6      while (true) {
7          while (new->priority > curr->priority) { //high num is low prio
8              prev = &curr->next;
9              curr = curr->next;
10         }
11         // insert node between prev and curr
12         new->next = curr;
13         if (prev == head)
14             ok = UpdateHead (head, curr, new, new->priority);
15         else
16             ok = CAS(prev, curr, new);
17         if (ok)
18             return; // success
19 }   }
```

(a) Enqueue

```
20 void* dequeue( int LogQueID) {
21     // " first " will get information on the node to dequeue:
22     // VA of its physical queue, and a pointer to the node
23     struct {
24         Node **physqueue;
25         Node* head;
26     } first ;
27     PrevFailed? = no;
28     while (true) {
29         first = PickHead (LogQueID,PrevFailed?); // return node
30         if (first.head == NULL)
31             return NULL; // empty
32         Node *next = first.head->next;
33         if (UpdateHead (first.physqueue, first.head,
34                         next, next->priority))
35             return first.head;
36         PrevFailed? = yes;
37 }   }
```

(b) Dequeue

Figure 3.3: Code to enqueue (a) and dequeue (b) a node with SNUG.

PickHead was described in Section 3.2. It takes as input operands the ID of a logical queue and a Boolean that indicates whether the CAS on the node previously provided by *PickHead* failed. The output operands are the VA of the chosen physical queue to dequeue from, and a pointer to the (expected) first node in that queue.

UpdateHead performs a CAS to modify a *Work Register* and, hence, to change the head of the corresponding physical queue. It changes the two fields of the *Work Register*—i.e., the pointer to the head node and its priority—atomically. It is used in both the enqueue and dequeue PQ library operations.

UpdateHead takes four input and one output operands (Table 3.1). The first input is the VA of the physical queue. The second input is the expected value of the pointer to the node at the head of the queue. The third and fourth inputs contain information about the node that the instruction wants to place at the head of the queue: a pointer to it and its priority. If this instruction succeeds, both pointer and priority in the *Work Register* are updated; if it fails, no change is made. The output is a Boolean with the outcome of the operation.

This operation is performed in a CAS hardware unit in the directory controller. In this way, operands do not have to flow from the directory controller to the core and back. A similar unit is provided in current GPUs to perform CASes in the cache hierarchy [24, 25]. Also note that, with this support, SNUG can perform arbitrary writes to the queue head.

FetchHead reads the pointer field of a *Work Register* and, thus, obtains the

head of a physical queue. It is used as part of the enqueue library operation, which needs to check the current value of the queue head to be able to change it with *UpdateHead*. *FetchHead* takes an input operand with the VA of the physical queue, and an output operand that receives the pointer to the node at the head of the queue.

3.4 Enqueue and Dequeue Operations

The previous instructions are not typically used directly by programmers. Instead, they are used in the PQ library to allocate queues and to perform the node enqueue and dequeue operations. Figures 3.3(a) and (b) show pseudo-code for the routines that enqueue and dequeue a node, respectively. Note that the programs that call the PQ library do not know about physical queues; they only reference logical queues.

For simplicity, Figures 3.3(a) and (b) focus on the physical deletions. We omit the details of handling logical deletions (Section 2.2), which are orthogonal to SNUG. Such details are discussed in Appendix A. In addition, the figures use simple linked lists. In practice, high-performance concurrent PQ libraries use the more efficient skip list [10]. Appendix B discusses how SNUG supports skip lists.

The enqueue routine (Figure 3.3(a)) is called with the ID of a logical queue and a node to *enqueue*. The routine first determines the VA of the local physical queue (Line 2); this is the queue where the node will be enqueued. The routine then uses *FetchHead* to read the pointer to the node at the head of the queue (Line 5). It then follows the linked list of nodes, reading the priority of each node, to find the place to insert the new node (Lines 7-10). If the node needs to be placed at the head, it uses *UpdateHead* to do so and fill the *Work Register* (Line 14). Otherwise, it uses a plain CAS (Line 16). Either *UpdateHead* or CAS can fail; if it does, the routine goes back to walking the queue to find where to enqueue.

The dequeue routine (Figure 3.3(b)) is called with the ID of the logical queue with the node to be dequeued. It returns one of the highest-priority nodes from the logical queue (not necessarily the highest one), by dequeueing it from the appropriate physical queue. The routine first uses *PickHead* with a clear *PrevFailed?* flag to find the VA of the physical queue and a pointer

to the node (Line 29). Then, the routine tries to dequeue the node using *UpdateHead* on the appropriate physical queue (Line 33). If *UpdateHead* succeeds, the routine returns a pointer to the dequeued node. Otherwise, the *PrevFailed?* flag is set and the routine repeats the use of *PickHead* and *UpdateHead* until the dequeue succeeds.

CHAPTER 4

DETAILED ASPECTS OF SNUG DESIGN

4.1 Algorithm to Pick the Node to Process

SNUG's algorithm for picking the next node to dequeue and process maintains a balance between two objectives. On one hand, SNUG aims to observe globally up-to-date information about the PQ, so that it can pick a high-priority node and minimize wasted work. On the other hand, SNUG also aims to avoid generating excessive network traffic, which would be the case if all *PickHead* invocations visited all the *Work Registers*. As per Section 3.2, SNUG achieves this balance by complementing the use of global accesses with two techniques. First, it *reuses* information in the *Snapshot Memory* provided by a global access to satisfy a few subsequent *PickHead* executions. Second, it *interleaves* several local queue accesses in between two consecutive global accesses.

The Decision Logic unit in the *PickHead* module (Figure 3.2) divides the stream of *PickHead* instruction executions into *phases*. Each phase begins with *PickHead* instruction that performs a global access, followed by U executions of *PickHead* that reuse the snapshot, and finally L executions of *PickHead* that generate requests to nearby queues.

SNUG reuses the obtained snapshot to identify additional high-priority nodes without generating additional network traffic. Recall that the snapshot contains many nodes, each of which was at the head of a physical queue at the time of the sample. These nodes are typically good candidates for processing. Eventually, however, the snapshot data becomes stale, as other processors dequeue items from the various queues. Therefore, after U *PickHead* executions, SNUG discards the snapshot.

SNUG then switches to visiting only nearby queues for L times, without obtaining a global snapshot. This trades off the quality of the executed work

for short periods, in order to avoid excessive network traffic. Specifically, SNUG identifies a few local queues that, according to the current data in the *Snapshot Memory*, contain nodes. In the next L *PickHead* executions, SNUG repeatedly accesses these local queues, uses the Sorter and Selector module to sort the responses, and returns to the software the highest-priority node among them. Following these L local accesses, a new phase begins, leading to the next *PickHead* triggering a global access.

4.2 Sorting the Nodes

After the Snapshot Memory receives a batch of 3-tuples from a set of queues, the Sorter & Selector module sorts these tuples in the order of decreasing priority. This is done using a hardware sorting network. Specifically, the module uses a Bitonic sorting network [26] for its low time and space overheads. Given n tuples, the network consists of $\log n \times (\log n + 1)/2$ sorting stages, and uses $n \times \log n \times (\log n + 1)/4$ comparator units. Each stage has $n/2$ comparator units that work concurrently, giving the overall network its speed.

In a global access, the Sorter & Selector module reads a register that contains the current time and performs a modulo operation with the current value of R. The result is an index that the module uses to select one tuple in the sorted array of tuples. In a local access, the Sorter & Selector module simply selects the highest-priority tuple. In either case, the tuple's physical queue and node pointer are passed to the software. The sorted list of tuples is kept latched in the Sorter & Selector module so that it can be reused in future *PickHead* executions.

4.2.1 Setting the Relaxation Count

The *Relaxation Count* (R) is a parameter that determines the quality of work for the first *PickHead* in a phase. It is defined as the maximum number of sorted nodes in the Sorter & Selector module from which one will be returned to the software. If R is too high, it may induce wasted work; if it is too low, it may cause *UpdateHead* failures in the dequeue routine (Figure 3.3(b)). To set R, we use real-time feedback from the application on how frequently

these *UpdateHead* operations fail. Since only the first *PickHead* in a phase performs a global access to create a fresh snapshot, we use the failure rate of only the *UpdateHead* operation immediately following the first *PickHead* to change R. Modified in this way, the value of R serves as an indicator of the global contention in the system at any time.

While the optimal R is likely to change across the execution time of an application, it is important that its value be impervious to short-term fluctuations of the *UpdateHead* failure rate, and instead reflect long-term trends or cycles. For this reason, we use the Exponential Moving Average (EMA) [27], a widely used indicator in statistical technical analysis.

As shown in Figure 3.2, the *PickHead* module includes a Double Exponential Moving Average (DEMA) unit, which receives information from the Decision Logic on how frequently *UpdateHead* failures occur for the first *PickHead* in a phase. The information is comprised of a single bit—1 for failure, 0 for success. Intuitively, observing many occurrences of 1 suggests that R is too small, while observing many 0 values suggests it is too high.

We define a *segment* as a series of bits of the same value (either 1 or 0) passed by the Decision Logic to the DEMA unit. On a segment termination, the DEMA unit uses the length of the segment to compute the following:

$$EMA = \alpha * sign * length + (1 - \alpha) * EMA$$
$$DEMA = \beta * EMA + (1 - \beta) * DEMA$$

where *length* is the number of entries in the segment, and α and β are constant parameters. The variable *sign* is 1 for a segment of 1s and -1 for a segment of 0s. With this setup, *UpdateHead* failures tend to push the DEMA slowly in the positive direction, while a string of *UpdateHead* successes move it in the opposite direction. Note that the DEMA changes only slowly.

When the application starts, the DEMA unit uses a default initial value R_0. During execution, the unit divides the time into *windows* of fixed size, and keeps calculating the DEMA. It keeps a positive DEMA threshold (T_{pos}) and a negative one (T_{neg}). If the DEMA has been above T_{pos} anytime in each of the previous two windows, the DEMA unit increases R one notch; if the DEMA has been below T_{neg} anytime in each of the previous two windows, R decreases by one notch. Figure 4.1 shows an example of how this algorithm works.

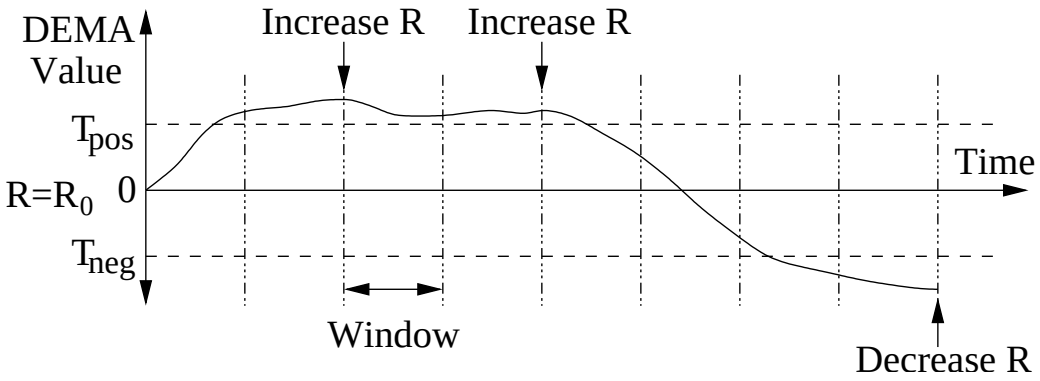


Figure 4.1: Example of how SNUG sets R.

Since each core computes its DEMA value independently, each core also modulates its own R independently. Interestingly, relaxation by one core has the serendipitous effect of reducing contention for sibling cores. As such, in most scenarios, some cores do not need to increase R as much as others, and can converge on a smaller R value.

Overall, R changes as the application executes different sections, and as different applications execute. However, this process is invisible to the programmer. Finally, after a context switch, R is reset to R_0.

CHAPTER 5

EVALUATION

5.1 Evaluation Environment

We perform our evaluation with cycle-accurate simulation of a 64-core chip using the GEM5 simulator [28]. Table 5.1 shows the baseline architecture modeled. We model a two-level hierarchical network design with clustering—8 cores share a single L2, and the 8 cache-coherent L2s are connected to a shared, 8-banked L3 with a crossbar.

Table 5.1: Parameters of the architecture evaluated.

Parameter	Value
Architecture	64 cores on chip
Core	2 GHz, single-issue
Private L1 Caches	32KB-I, 32KB-D WB, 8-way, 2 cycles hit latency
Per-cluster L2 Cache	1MB WB, 16-way, 12 cycles hit latency
Shared L3	16MB WB, 8 banks, 16-way, 30 cycles hit latency
Cache line size	64B
Coherence	Two-level MOESI directory protocol
Network	32B wide, hierarchical, crossbar with snoop filters
Main memory	$\approx$ 200 cycles
SNUG Parameters	
Snapshot reuses (U)	4
Local accesses (L)	4
R_0, Window	32, 100K cycles
DEMA thresholds	T_{pos} = 5, T_{neg} = -2.5
EMA, DEMA constants	α = 0.6, β = 0.6
Bitonic sorting network delay	21 cycles

We compare the following concurrent priority queue (PQ) implementations:

Centralized skip list (*SW-SK*). This skip list implementation is based on the algorithm outlined in [20]. The levels for new skipped nodes are chosen based on a geometric distribution. The maximum number of levels is 24.

Centralized SprayList (*SW-SP*). This SprayList builds on top of the skip list by spraying the Pops over a range of starting nodes in the list. We

use the SprayList parameters as advised by the authors in [12]. The spray is started at the height of $\lfloor \log_2 t \rfloor + 1$, and the jump length at each level is $\lfloor \log_2 t \rfloor + 1$, where t is the number of threads. The number of levels to descent between jumps is 1, and the maximum number of levels is 24.

Distributed software (*SW-D*). This is a distributed software PQ with a per-core skip list. Threads always enqueue to their local skip list. For dequeue, the local skip list is tried first, failing which the thread attempts to steal work from nearby skip lists.

Centralized hardware (*HW-C*). This design is a centralized version of Snug. It consists of a single shared skip list with a single, centralized *Work Register*. It uses the *FetchHead* and *UpdateHead* instructions to access the *Work Register* (Section 3.3). A single *Work Register* obviates the need for a *PickHead* instruction.

Distributed hardware (*HW-D*). This is Snug. It uses distributed skip lists, each of which is supported by a *Work Register* at the directories. Threads always enqueue to their local queue. For dequeue, a *PickHead* instruction returns the node to dequeue as described in Section 4.1. For the L local accesses, we use the following algorithm. A *PickHead* visits only a *single* queue. By default, such a queue is the local one. However, if the data in the *Snapshot Memory* indicates that the local queue is empty, instead *PickHead* randomly visits one of the queues that the *Snapshot Memory* indicates is not empty. Table 5.1 outlines Snug's parameters.

Finally, we also evaluate a software version of Snug, where global dequeues scan all the heads of the distributed queues in software. This PQ is an order of magnitude slower than *SW-D*, due to the overhead of the serially scanning of the queue heads. Therefore, we omit it from the evaluation.

5.2 Characterizing Applications

We evaluate Snug on three applications which use concurrent PQs, two from graph analytics and an event-driven simulation.

Graph analytics. The Single-Source Shortest-Paths (SSSP) application uses Dijkstra's algorithm [18] to compute the shortest distance to all graph nodes, starting from a source node. Parallelism is exposed by relaxing nodes

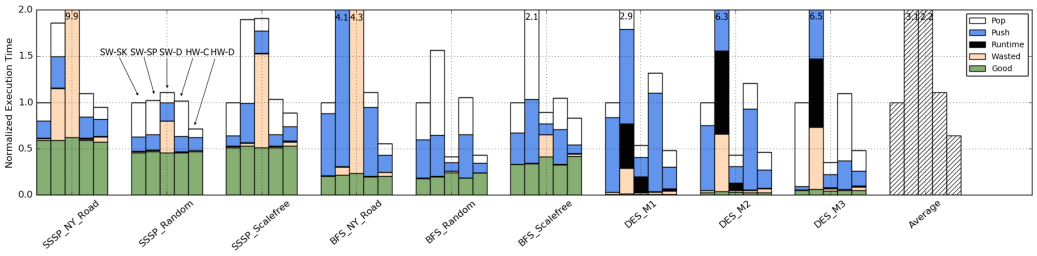


Figure 5.1: Execution time of the evaluated applications and inputs on different priority queue implementations.

out of priority order (Section 2.1). Out-of-order scheduling could lead to multiple, redundant relaxations of some nodes, creating *wasted work*. This could increase the convergence time of the algorithm, though correctness is never violated. We base our SSSP implementation on the Push-operator [2], since we found it to constantly outperform the pull-operator based approach. Breadth-First Search (BFS) uses the same algorithm as SSSP, but the weight of all the edges of the graph is 1. As we show in Section 5.3, using unit-edge weights drastically changes how the application responds to the various priority queue schedulers.

Event-driven simulation. Discrete Event Simulation (DES) is a system-modeling program with wide variety of use cases [29, 30]. The progress of the system is determined by the execution of events, some of which have dependence on other events, thereby creating a partial order. For our implementation, we generate a matrix of dependencies between events based on the approach outlined in [12]. For an event, the number of events that are dependent on it is distributed geometrically with δ, and the mean distance between the source and dependent node is parameterized with γ. We synthesize three input matrices for the DES application with a fixed δ and different values of γ. To load-balance the distributed queue variants (*SW-D* and *HW-D*), events are inserted into all the queues in a round-robin fashion at the beginning of the program. When a thread dequeues an event, it checks if all the dependencies for that event have been executed. If not, the event is enqueued again; otherwise it is executed.

The input graphs for SSSP and BFS, and the input matrices for DES are shown in Table 5.2. All runs in the evaluation are with 64 threads, each pinned to one core. For the distributed PQs, 64 lists are instantiated, one

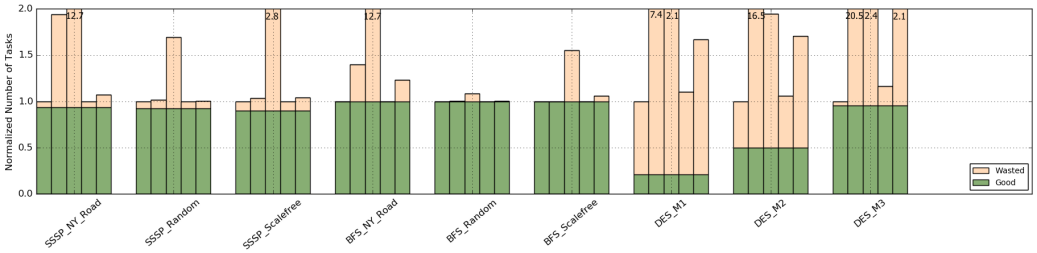


Figure 5.2: Breakdown of the tasks in the applications into good and wasted tasks.

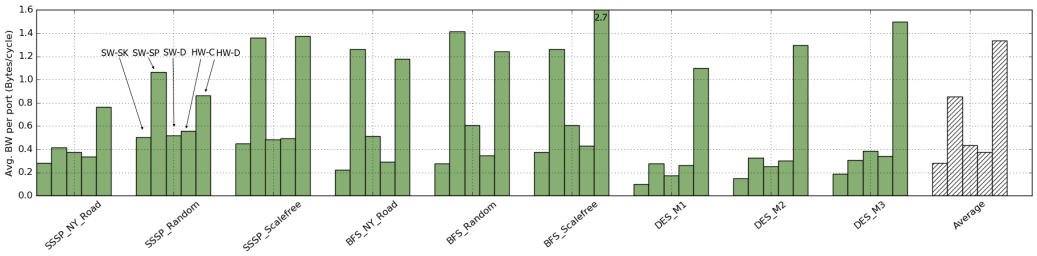


Figure 5.3: Average bandwidth consumption in each of the global crossbar ports connected to the core clusters.

local to each thread. For *HW-D*, 64 *Work Registers* are used. The execution time measurements are for the region of interest, expunging the graph or matrix load time.

Table 5.2: Program inputs.

Input	Description
Graphs:	
NY Road	Road network graph for New York City with edge weights as the travel time [31]. —V— = 264,346, —E— = 733,846
Scalefree	Graph with degree distribution that follows the power law, generated with R-MAT [32, 2]. —V— = 260,237, —E— = 2,097,152
Random	Connected graph with random edges. —V— = 100,000, —E— = 400,000
Matrices:	
M_1	Number of events = 50K, $\delta = 10$, $\gamma = 20$
M_2	Number of events = 50K, $\delta = 10$, $\gamma = 100$
M_3	Number of events = 50K, $\delta = 10$, $\gamma = 200$

To understand the behavior of the different PQ implementations, we recall how our applications work. In the applications, each thread dequeues a single unit of work from the PQ, processes it, and (possibly) enqueues one or more units of work to the PQ. This process repeats in a loop.

We categorize the tasks based on the kind of work that was done in it—

good tasks for good work, wasted tasks for wasted work. For SSSP and BFS, good work consists of graph edge relaxations that update the label of a node to its true distance (see Section 2.1), and wasted work consists of all the remaining, redundant relaxations. Wasted work occurs either due to thread concurrency and, most notably, relaxed PQ semantics. For DES, good work consists of dequeueing an event after all of its dependencies have executed, thereby enabling its own execution. Otherwise, the event is inserted back into the PQ for later processing, and the task is classified as wasted.

5.3 Execution Time

Figure 5.1 compares the execution time of the applications and inputs under the different PQ implementations. For each application and input, we show, from left to right, *SW-SK*, *SW-SP*, *SW-D*, *HW-C*, and *HW-D*, all normalized to *SW-SK*. The time is broken down into *Pop* synchronization (the time spent in PQ dequeue operations), *Push* synchronization (the time spent in PQ enqueue operations), good work, wasted work, and runtime (which is the execution of auxiliary code, such as thread termination).

To understand these bars, Figure 5.2 provides a breakdown of the total tasks executed in the program classified into good and wasted tasks. The total number of tasks is normalized to the number in *SW-SK*. Note that wasted tasks affect the execution in two ways. First, they induce redundant processing of graph edges or events. These cycles appear as wasted work in Figure 5.1. Second, they cause extraneous enqueues and dequeues to the PQ, thereby increasing synchronization time. This causes an increase in the Push and Pop times in Figure 5.1.

In the following, we consider all the PQ implementations in detail except for *HW-C*. It differs from *SW-SK* mostly by having a single *Work Register* which points to the highest-priority task. In general, its overall performance is similar to *SW-SK*. In some cases, it performs slightly worse than *SW-SK* due to the extra overheads of the new instructions.

SSSP. As shown in Figure 5.1, *HW-D* performs well in SSSP. On average across the three inputs, it reduces the execution time by 1.2×, 1.5×, and 4.7× compared to *SW-SK*, *SW-SP*, and *SW-D*, respectively. A comparison

between *HW-D* and the state-of-the-art *SW-SK* shows the advantages of *HW-D*. *HW-D* reduces the Pop time substantially, while only increasing the Push time slightly. *HW-D* reduces Pop time because it eliminates the contention on the queue head in *SW-SK* with the use of distributed queues and the *PickHead* instruction, avoiding a central point of contention. *HW-D*'s Push time is slightly higher than *SW-SK* because *HW-D* is more relaxed, and it ends up creating slightly more bad work (Figure 5.2) and enqueuing more tasks.

HW-D performs much better than *SW-SP* and *SW-D* because these two PQ implementations spend substantial time in wasted work—especially on the NY-Road input. *SW-SP* returns tasks from among the first $O(t \log^3 t)$ ones in the PQ (with high probability), where t is the number of threads (Section 2.1). For sparse graphs like NY-Road, the number of tasks in the PQ is small. This causes *SW-SP* to return a random task, ignoring the priorities altogether, and leads to wasted work. Moreover, in a small PQ, the *SW-SP*'s random walks often reach the end of the list. When this happens, *SW-SP* subsequently performs a linear scan of the PQ, to make sure no work was missed [12], and this increases Pop time.

Similarly, *SW-D* has significant wasted work. This is because each thread takes tasks from its own queue, without looking for the highest-priority task. The lack of synchronization overheads in *SW-D* is unable to compensate for this wasted work.

BFS. *HW-D* performs even better in BFS, reducing the average execution time by 1.8×, 4.5×, and 3.3× compared to *SW-SK*, *SW-SP*, and *SW-D*, respectively (Figure 5.1). In BFS, all the edges of the graph have a weight of 1. This increases the available parallelism, since there are now multiple paths of the same distance from the source node to each destination. This changes the behavior of the PQs implementations.

HW-D attains better performance than the centralized PQs (*SW-SK* and *SW-SP*) because of the reduced synchronization. Specifically, Pop time is lower because *HW-D* distributes the dequeues across many queues, and Push time is lower because the enqueues traverse shorter per-core queues.

HW-D is also faster than *SW-D* because the latter still has substantial wasted work in the NY-Road input. This can be seen both in Figure 5.2 and in Figure 5.1. As each thread takes tasks from its own queue, it is likely to

find low-priority tasks. The other inputs (Random and Scalefree) are denser graphs, which result in much less wasted work.

DES. Finally, in DES, *HW-D* reduces the average execution time by 2.1× and 11× compared to *SW-SK* and *SW-SP*, respectively. However, it is on average slightly slower than *SW-D* (Figure 5.1). To understand this behavior, we note that, in DES, it is easy to generate wasted tasks (Figure 5.2). Parallel threads dequeue simulation events before all of their dependencies are executed, which produces a large number of wasted tasks. Threads spend substantial time dequeuing and enqueuing wasted tasks. The work done per wasted task, however, is tiny.

In this environment, centralized PQ implementations like *SW-SK* and *SW-SP* suffer from significant synchronization time. The problem is especially acute in *SW-SP* due to the previously discussed issue of a sub-optimal spray. Therefore, these two implementations are not competitive.

Since *SW-D* is more relaxed than *HW-D*, it executes more wasted tasks than *HW-D* (Figure 5.2). However, since the tasks are very small, the actual wasted work time is small and does not hurt *SW-D*'s execution time (Figure 5.1). Since, in addition, synchronization is very cheap in *SW-D*, *SW-D* ends up having an execution time that is on average 8% lower than *HW-D*.

Overall, we see that several factors determine the performance of PQ implementations. However, averaged across all applications and inputs, Snug's *HW-D* design in Figure 5.1 reduces the execution time by 1.6×, 4.9×, and 3.4× compared to the state-of-the-art skip list, SprayList, and software distributed PQs, respectively. Compared to the latter two relaxed PQ designs, it can be shown that *HW-D* reduces the number of wasted tasks by 4.4× and 18.2×, respectively.

Snug attempts to observe globally up-to-date information about the PQ while avoiding the creation of excessive network traffic due to accesses to the *Work Registers*. To assess the resulting traffic, we measure the bandwidth consumption in the ports of the global crossbar that connect to the core clusters. Figure 5.3 shows the average bandwidth consumption in each of those ports of the global crossbar in bytes per cycle. We consider all the network traffic, including the traffic that is unrelated to PQ accesses. We show data for all the PQ implementations, applications, and inputs.

We see that, on average, the centralized PQs (*SW-SK* and *HW-C*) and the

software distributed one (*SW-D*) consume the least amount of bandwidth. Out of all the PQ implementations, Snug consumes the most bandwidth on average—about 4.7× compared to *SW-SK*, and 3.5× compared to the best-performing alternative design on each application, as shown in Figure 5.1. Crucially, however, Snug's bandwidth consumption is modest in absolute terms. On average across all applications, Snug consumes 1.33 bytes per cycle, or 4.1% of the crossbar's 32 bytes per cycle link capacity, which is the same link capacity assumed in related work [33] and deployed in commercial processors such as Intel's Haswell.

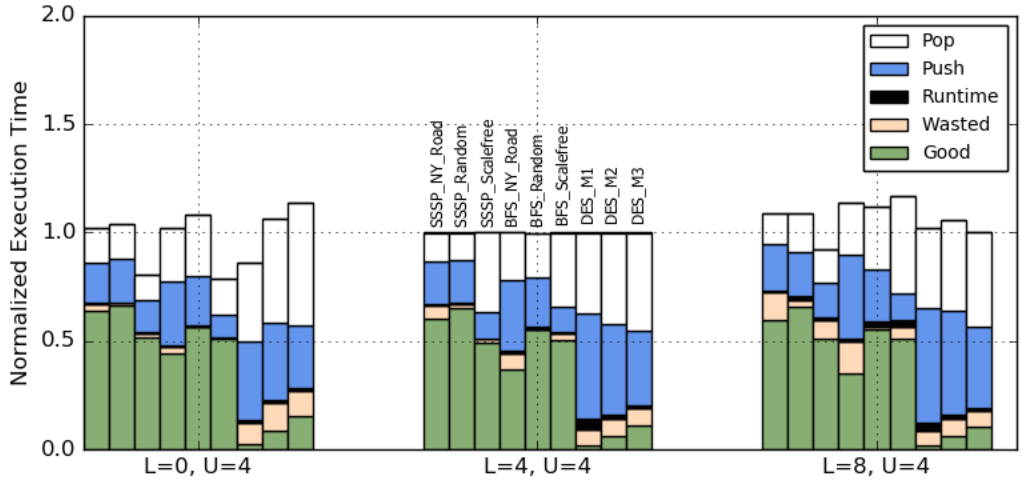
5.4 Sensitivity Analysis

The algorithm to pick the next node to process (Section 4.1) has two main parameters, namely the number of local accesses (L), and the number of snapshot reuses (U). Our Snug design sets both parameters to 4. In this section, we change these values one at a time.

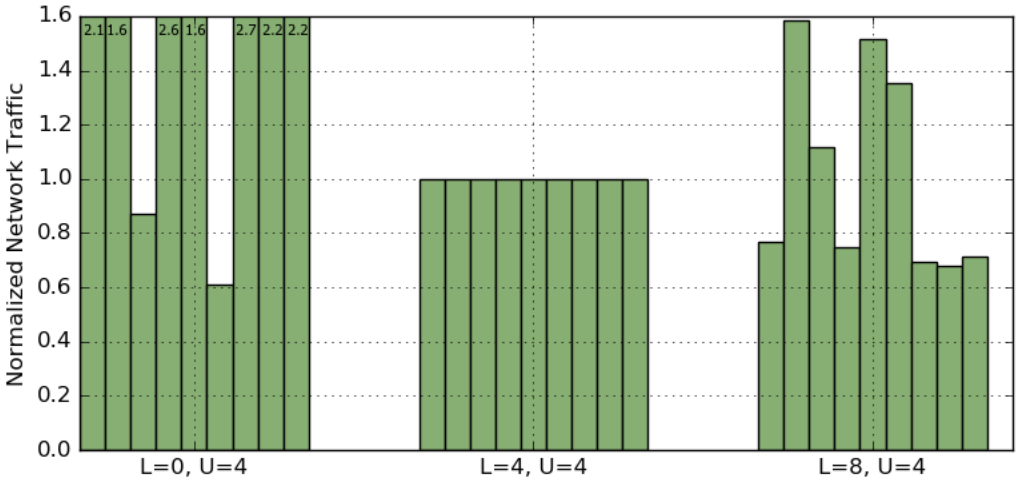
Figure 5.4 shows the sensitivity of Snug's execution time and network traffic to changes in the L parameter, as we vary it from 0 to 4 and 8. In all cases, U is kept at 4. For each setting, the figure shows a bar for each application and input data. For a given application and input, the bars are normalized to the ($L = 4$, $U = 4$) setting. The execution time bars are broken down into the usual categories.

We see that the performance of Snug is sensitive to L. When $L = 0$, Snug dequeues better tasks, since each *PickHead* creates a new snapshot and selects a node from the top R nodes. However, this design creates higher traffic in the network, due to the higher frequency of global accesses. As a result, while the execution time is lower for some cases, it is higher for others. Due to the high traffic, this configuration is not desirable. At the other end, when $L = 8$, Snug dequeues more low-priority tasks from the local queue. For some applications, this means more wasted work, and even more traffic. The execution time goes slightly up for most applications. Overall, the best design is for $L = 4$.

Figure 5.5 shows the sensitivity to changes in U. The charts are organized as in Figure 5.4. We see that the performance also changes with U. When $U = 0$, the traffic is higher because there is no reuse of the snapshot. This



(a) Execution time



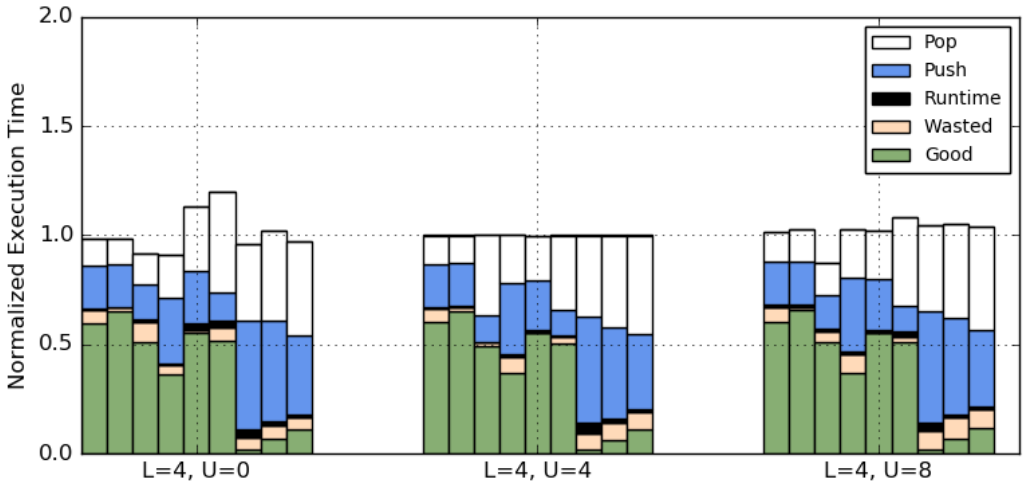
(b) Network traffic

Figure 5.4: Sensitivity to the number of local accesses (L).

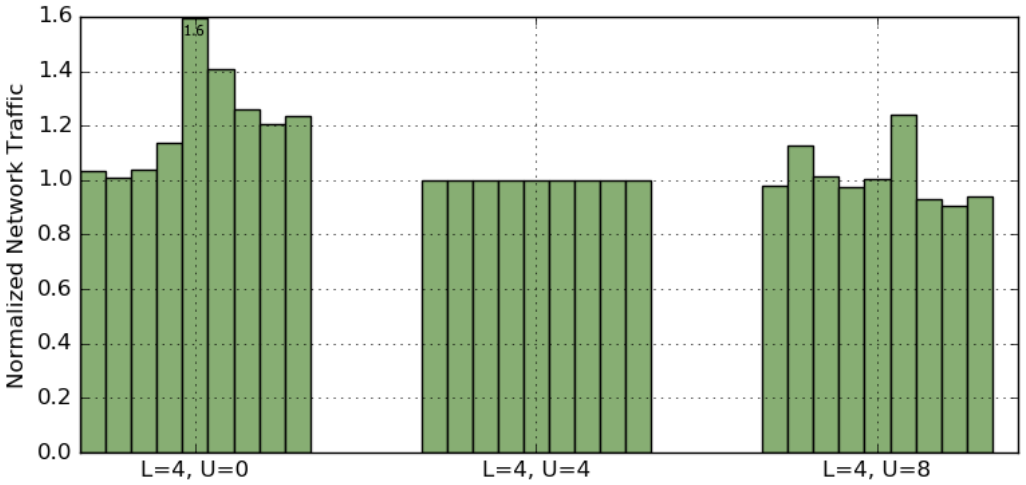
results in an increase in the execution time of several of the applications and inputs. On the other hand, when $U = 8$, the snapshot is reused past the point when the information is useful. As a result, the Pop time increases, producing a slightly higher execution time. In summary, $U = 4$ is the best design point.

5.5 Characterizing R Adaptivity

The *PickHead* module in each core adjusts the *Relaxation Count* R based on the rate of synchronization failures, which indicate the degree of contention.



(a) Execution time

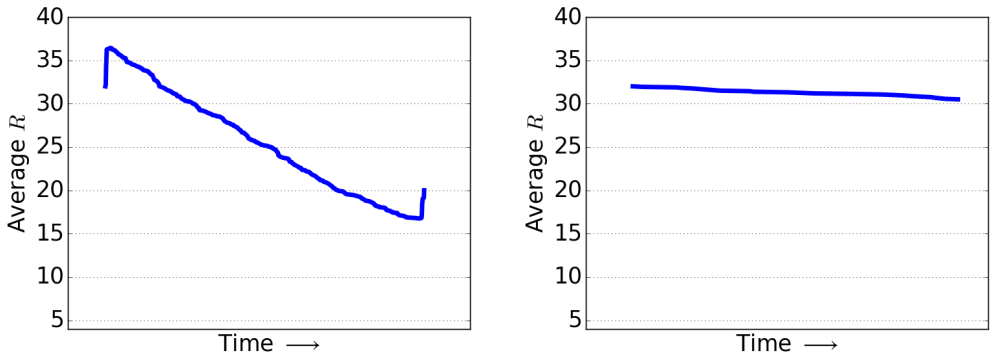


(b) Network traffic

Figure 5.5: Sensitivity to the number of reuses of a snapshot (U).

Figure 5.6 shows two representative examples of R's behavior as the application executes: SSSP on the NY-Road graph, and DES on the M_3 matrix. The X-axis is the normalized execution time. At each point in time, the figures show the average value of R across all cores. In both cases, we start with our default $R_0 = 32$.

SSSP on the NY-Road graph (Figure 5.6(a)) demonstrates how Snug adjusts R so that it picks the highest-quality tasks to execute without causing synchronization hotspots. When the application starts, there is only one task in one queue. Therefore, initially, all cores attempt to dequeue this task. Slowly, more work is being created and deposited in other queues. Overall, this is a period of high contention on the handful of queues that



(a) SSSP on NY-Road graph.

(b) DES on M_3 matrix.

Figure 5.6: Average value of the Relaxation count (R) across all cores as the applications progress.

contain tasks. As a result, R increases quickly from its initial value of 32.

Gradually, all queues obtain tasks. Since R is large enough to distribute the concurrent dequeues across the many queues, synchronizations succeed. This drives the DEMA value (Section 4.2.1) below the negative threshold, and triggers a decrease in R. The descent continues as long as the cores do not observe enough dequeue conflicts. The relatively slow rate of descent is affected by the fact that only global accesses affect the DEMA. Finally, as the application runs out of work and most queues become empty again, the contention increases. This causes the final spike in R.

DES on the M_3 matrix (Figure 5.6(b)) shows a different behavior, in which R remains close to its initial value. The difference in behavior from SSSP is due to two reasons. First, the section of execution that we evaluate starts after work has been distributed across all the queues. This eliminates the initial contention that lead R to increase in Figure 5.6(a). More importantly, however, is the fact that DES has tasks with very little work, and substantial contention. Thus, synchronization fails often, and the DEMA value rarely moves below the negative threshold. As a result, R decreases infrequently.

CHAPTER 6

RELATED WORK

The Galois [2] graph analytics system contains a priority-based scheduler named OBIM, which has been shown to outperform PQ designs based on Intel's TBB library [23]. Like relaxed PQs, OBIM may execute tasks out of priority order. OBIM is closely tied to the Galois system, requiring users to port applications to Galois in order to use it. Snug targets general data structure libraries.

Providing hardware support for queueing in a multiprocessor environment has received much interest [33, 34, 35, 36, 37]. Carbon [34] is an architecture for efficient task queueing and scheduling in hardware. It focuses on applications with fine-grained tasks. It provides cores with dedicated hardware queues and orchestrates the movement of tasks between the queues. ADM [36] accelerates fine-grained task scheduling by providing a per-core hardware messaging module. Bypassing the memory hierarchy, the module enables effective communication between threads, which manage task queues in software.

Swarm [33, 38] mines parallelism from sequential programs written in a task-based programming model. Swarm builds on prior TLS and HTM schemes. It executes tasks speculatively and out of order, and efficiently speculates thousands of tasks ahead of the earliest task to uncover ordered parallelism. In contrast, Snug targets parallel programs, which already explicitly synchronize, and thus has a simpler hardware mechanism that is not based on speculation. Swarm also supports task queueing in hardware. In Swarm, cores enqueue tasks to a randomly chosen remote queue and dequeue tasks from their local queue. In Snug, cores enqueue locally and dequeue either remotely or locally.

The CASPAR [39] architecture executes certain contending CAS instructions in parallel instead of serially. CASPAR applies to CASes in *enqueue* operations, whose written value is known early-on. In contrast, Snug ad-

dresses the contention in *dequeue* operations, to which CASPAR does not apply. Whereas CASPAR focuses on improving the scalability of contended CAS instructions, SNUG attempts to avoid contention altogether by relaxing the PQ data structure; SNUG thus leverages higher-level insights, about the compatibility of algorithms with relaxed PQs, rather than low-level insights about the CAS instruction.

CHAPTER 7

CONCLUSION

Current priority-based task scheduling algorithms pose an unfortunate trade-off: alleviating the synchronization hotspot by using relaxed PQs leads to wasted work. To avoid this trade-off, this thesis introduced the Snug architecture. Snug consists of per-core *Work Registers*, a node pick instruction and other ISA extensions, and a module for dynamically adapting the relaxation in task selection.

We evaluated Snug on a simulated 64-core chip. We compared the execution time of graph and discrete event simulation applications under Snug and several other PQ algorithms. Such algorithms included software-only PQs based on a skip list, a SprayList, and a distributed skip list; and a hardware-based centralized PQ. We found that Snug relaxes the priority order in a PQ algorithm by just the right amount, selecting high-quality tasks while avoiding hotspots, minimizing wasted work, and consuming acceptable network bandwidth. Snug reduces the average execution time of the applications by 1.6×, 4.9× and 3.4× compared to the state-of-the-art skip list, SprayList, and software-distributed PQs, respectively. Moreover, compared to the latter two relaxed PQ designs, Snug reduces the number of wasted tasks by 4.4× and 18.2×, respectively.

APPENDIX A

LOGICAL DELETIONS

Logical deletions in concurrent lists are required to maintain atomicity of deletions in the face of concurrent operations. Deleting a node x by updating only its predecessor's *next* pointer might undo the effect of a concurrent operation that updates x. To prevent such an atomicity violation, an operation should only update the predecessor's *next* pointer (physical deletion) after first *logically* deleting the target node. Logical deletion prevents concurrent operations from updating the node. The standard approach for logical deletions involves co-locating a flag bit with the LSB of the *next* field [21, 40]. Updates fail (and retry) if they observe that their target node has this flag bit set; this ensures that the CAS of an update succeeds only if the pointer it is updating has not been flagged since being read.

APPENDIX B

SUPPORTING SKIP LISTS

A skip list is a sorted linked list where individual nodes are linked into multiple sorted lists, in an effort to speed-up searches by "skipping over" irrelevant nodes. In a skip list, a node has an array of *next* pointers, not just one, linking it into multiple lists. Thanks to skipping, a skip list search traverses n nodes in $O(\log n)$ time (expected). Concurrent skip list algorithms support scalable read-mostly searches and insertions, and can access the highest-priority node in $O(1)$ time [19, 41, 42, 11]. Concurrent skip lists are a popular choice for concurrent PQs [6, 8, 10, 11]. SNUG's mechanisms are compatible with a skip list-based PQ. In this case, the *Work Register* stores the head of the bottom-level list. The heads of the other, higher-level lists, remain located in memory and are maintained by the software PQ library.

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[2] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP*, 2013.

[3] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, no. 6, 1957.

[4] R. M. Fujimoto, "Parallel discrete event simulation," *CACM*, vol. 33, no. 10, pp. 30–53, 1990. [Online]. Available: http://doi.acm.org/10.1145/84537.84545

[5] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[6] I. Calciu, H. Mendes, and M. Herlihy, "The adaptive priority queue with elimination and combining," in *DISC*, 2014.

[7] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott, "An efficient algorithm for concurrent priority queue heaps," *IPL*, vol. 60, no. 3, pp. 151–157, 1996.

[8] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *OPODIS*, 2013.

[9] Y. Liu and M. Spear, "Mounds: Array-based concurrent priority queues," in *ICPP*, 2012.

[10] I. Lotan and N. Shavit, "Skiplist-based concurrent priority queues," in *IPDPS*, 2000.

[11] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *JPDC*, vol. 65, no. 5, pp. 609–627, 2005.

[12] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A scalable relaxed priority queue," in *PPoPP*, 2015.

[13] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: MultiQueues: Simple relaxed concurrent priority queues," in *SPAA*, 2015.

[14] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The Lock-free k-LSM relaxed priority queue (Poster)," in *PPoPP*, 2015.

[15] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas, "Data structures for task-based priority scheduling (poster)," in *PPoPP*, 2014.

[16] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *PPoPP '12*, 2012.

[17] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, 2001.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[19] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory, February 2004.

[20] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *CACM*, vol. 33, no. 6, pp. 668–676, June 1990.

[21] T. Harris, "A pragmatic implementation of non-blocking linked-lists," in *DISC*, 2001.

[22] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel asynchronous label-correcting methods for shortest paths," *Journal of Optimization Theory and Applications*, vol. 88, no. 2, Feb. 1996.

[23] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par*, 2015.

[24] "NVIDIA corp. NVIDIA CUDA programming guide v3.1," http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[25] "Khronos group. opencl," http://www.khronos.org/opencl/.

[26] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121 pp. 307–314.

[27] "Moving average," https://en.wikipedia.org/wiki/Moving_average.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[29] J. Karnon, J. Stahl, A. Brennan, J. J. Caro, J. Mar, and J. Möller, "Modeling using discrete event simulation a report of the ISPOR-SMDM modeling good research practices task force–4," *Medical Decision Making*, vol. 32, no. 5, pp. 701–711, 2012.

[30] M. Holly and C. Tropper, "Parallel discrete event n-body dynamics," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: http://dx.doi.org/10.1109/PADS.2011.5936760 pp. 1–10.

[31] "9th DIMACS implementation challenge," http://www.dis.uniroma1.it/challenge9/download.shtml.

[32] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." SIAM.

[33] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *MICRO*, 2015.

[34] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 162–173, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273440.1250683

[35] B. Khan, D. Goodman, S. Khan, W. Toms, P. Faraboschi, M. Luján, and I. Watson, "Architectural support for task scheduling: Hardware scheduling for dataflow on NUMA systems," *J. Supercomput.*, vol. 71, no. 6, pp. 2309–2338, June 2015. [Online]. Available: http://dx.doi.org/10.1007/s11227-015-1383-2

[36] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736055 pp. 311–322.

[37] G. Al-Kadi and A. S. Terechko, "A hardware task scheduler for embedded video processing," in *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 2009, pp. 140–152.

[38] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *MICRO*, 2016.

[39] T. Gangwani, A. Morrison, and J. Torrellas, "CASPAR: Breaking serialization in lock-free multicore synchronization," in *ASPLOS*, 2016.

[40] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA*, 2002.

[41] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skip list algorithm," in *SIROCCO*, 2007.

[42] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.